Pyjnius Documentation

Release 1.0a1

Kivy Team and other contributors

May 13, 2023

Contents

Pyjnius is a Python library for accessing Java classes.

It either starts a new JVM inside the process, or retrieves the already surrounding JVM (for example on Android).

This documentation is divided into differents parts. We recommend you to start with *Installation*, and then head over to the *Quickstart*. You can also check *Android* for specific example for the Android platform. If you'd rather dive into the internals of Pyjnius, check out the *API* documentation.

CHAPTER 1

Installation

PyJNIus depends on Cython and the Java Development Kit (includes the Java Runtime Environment).

1.1 Installation on GNU/Linux distributions

You need the GNU Compiler Collection (GCC), the JDK and JRE installed (openjdk will do), and Cython. Then, just type:

```
sudo python setup.py install
```

If you want to compile the extension within the directory for any development, just type:

```
make
```

You can run the tests suite to make sure everything is running right:

```
make tests
```

1.2 Installation for Android

To use pyjnius in an Android app, you must include it in your compiled Python distribution. This is done automatically if you build a Kivy app, but you can also add it to your requirements explicitly as follows.

If you use buildozer, add pyjnius to your requirements in buildozer.spec:

```
requirements = pyjnius
```

If you use python-for-android directly, add pyjnius to the requirements argument when creating a dist or apk:

```
p4a apk --requirements=pyjnius
```

1.3 Installation for Windows

Python and pip must be installed and present in the `PATH` environment variable.

1. Download and install the JDK containing the JRE:

   http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Edit your system and environment variables (use the appropriate Java bitness and version in the paths):

   Add to your Environment Variables:

   - `JAVA_HOME`: C:\Program Files\Java\jdk1.7.0_79\bin
   - `PATH`: C:\Program Files\Java\jdk1.7.0_79\jre\bin\server contains the `jvm.dll` necessary for importing and using PyJNIus.

   Note: set PATH=%PATH%;C:\Program Files\Java\jdk1.7.0_79\jre\bin\server

   Add to System Variables or have it present in your `PATH`:

   - `PATH`: C:\Program Files\Java\jdk1.7.0_79\bin‘

3. Download and install the C compiler:

   a) Microsoft Visual C++ Compiler for Python 2.7:

   http://aka.ms/vcpython27

   b) MinGWPy for Python 2.7:

   https://anaconda.org/carlkl/mingwpy

   c) Microsoft Visual C++ Build Tools (command-line tools subset of Visual Studio) for Python 3.5 and 3.6:

   https://visualstudio.microsoft.com/downloads/

   For other versions see Python’s Windows Compilers wiki.

4. Update pip and setuptools:

```
python -m pip install --upgrade pip setuptools
```

5. Install Cython:

```
python -m pip install --upgrade cython
```

6. Install Pyjnius:

```
pip install pyjnius
```

Note: In case of MinGWPy’s GCC returning a `CreateProcess failed: 5` error you need to run the command prompt with elevated permissions, so that the compiler can access the JDK in `C:\Program Files\Java\jdkx.y.z_b` or `C:\Program Files (x86)\Java\jdkx.y.z_b`.

1.4 Installation for macOS

Python and pip must be installed and present in the `PATH` environment variable.

1. Download and install the JDK containing the JRE:

   http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Edit your system and environment variables (use the appropriate Java bitness and version in the paths):

   Add to your Environment Variables:

   - `export JAVA_HOME=/usr/libexec/java_home`

3. Install Xcode command-line tools.

4. Update pip and setuptools:

```
python -m pip install --upgrade pip setuptools
```

5. Install Cython:

```
python -m pip install --upgrade cython
```

6. Install Pyjnius:

```
pip install pyjnius
```

1.5 Installation for Conda

Similar to PIP there is a package manager for *Anaconda <https://www.anaconda.com/what-is-anaconda/>* called Conda. An unofficial compiled distributions of PyJNIus for Conda supported platforms you can find at https://anaconda.org/conda-forge/pyjnius.

You can install `pyjnius` with this command:

```
conda install -c conda-forge pyjnius
```

Or if you want a specific package label e.g. `gcc7`:

```
conda install -c conda-forge/label/gcc7 pyjnius
```

CHAPTER 2

Quickstart

Eager to get started? This page will give you a good introduction to Pyjnius. It assumes you have already Pyjnius installed. If you do not, head over the *Installation* section.

2.1 A minimal example

A minimal Pyjnius example looks something like this:

```
from jnius import autoclass

Stack = autoclass('java.util.Stack')
stack = Stack()
stack.push('hello')
stack.push('world')

print stack.pop() # --> 'world'
print stack.pop() # --> 'hello'
```

Just save it as *test.py* (or something similar) and run it with your Python interpreter. Make sure not to call your application *jnius.py* because it would conflict with Pyjnius itself:

```
$ python test.py
world
hello
```

To load nested java classes, use the "$" separator as so:

```
version = autoclass("android.os.Build$VERSION")
base_os = version.BASE_OS
```

2.2 Automatic recursive inspection

Pyjnius uses Java reflection to give you a new autoclass() if the return type is not a native type. Let's see this example:

```
System = autoclass('java.lang.System')
System.out.println('Hello World')
```

We only declared the first System class, but we are able to use all the static fields and methods naturally. Let's go deeper:

```
>>> System = autoclass('java.lang.System')
>>> System
<class 'jnius.java.lang.System'>
>>> System.out
<java.io.PrintStream at 0x234df50 jclass=java/io/PrintStream jself=37921360>
>>> System.out.println
<jnius.JavaMethodMultiple object at 0x236adb8>
```

The recursive reflection always gives you an appropriate object that reflects the returned Java object.

CHAPTER 3

Android

Android has a great and extensive API to control devices, your application etc. Some parts of the Android API are directly accessible with Pyjnius but some of them require you to code in Java.

Note: Since Android 8.0 (Oreo) the maximum limit for the local references previously known as "local reference table overflow" after 512 refs has been lifted, therefore PyJNIus can create proper Java applications with a lot of local references. Android JNI tips

3.1 Get the DPI

The DisplayMetrics contains multiple fields that can return a lot of information about the device's screen:

```
from jnius import autoclass
DisplayMetrics = autoclass('android.util.DisplayMetrics')
metrics = DisplayMetrics()
print('DPI', metrics.getDeviceDensity())
```

Note: To access nested classes, use *$* e.g. *autoclass('android.provider.MediaStore$Images$Media')*.

3.2 Recording an audio file

By looking at the Audio Capture guide for Android, you can see the simple steps for recording an audio file. Let's do it with Pyjnius:

```
from jnius import autoclass
from time import sleep
```

(continues on next page)

(continued from previous page)

```
# get the needed Java classes
MediaRecorder = autoclass('android.media.MediaRecorder')
AudioSource = autoclass('android.media.MediaRecorder$AudioSource')
OutputFormat = autoclass('android.media.MediaRecorder$OutputFormat')
AudioEncoder = autoclass('android.media.MediaRecorder$AudioEncoder')

# create out recorder
mRecorder = MediaRecorder()
mRecorder.setAudioSource(AudioSource.MIC)
mRecorder.setOutputFormat(OutputFormat.THREE_GPP)
mRecorder.setOutputFile('/sdcard/testrecorder.3gp')
mRecorder.setAudioEncoder(AudioEncoder.AMR_NB)
mRecorder.prepare()

# record 5 seconds
mRecorder.start()
sleep(5)
mRecorder.stop()
mRecorder.release()
```

And tada, you'll have a */sdcard/testrecorder.3gp* file!

3.3 Playing an audio file

Following the previous section on how to record an audio file, you can read it using the Android Media Player too:

```
from jnius import autoclass
from time import sleep

# get the MediaPlayer java class
MediaPlayer = autoclass('android.media.MediaPlayer')

# create our player
mPlayer = MediaPlayer()
mPlayer.setDataSource('/sdcard/testrecorder.3gp')
mPlayer.prepare()

# play
print('duration:', mPlayer.getDuration())
mPlayer.start()
print('current position:', mPlayer.getCurrentPosition())
sleep(5)

# then after the play:
mPlayer.release()
```

3.4 Accessing the Activity

This example will show how to start a new Intent. Be careful: some Intents require you to setup parts in the *AndroidManifest.xml* and have some actions performed within your Activity. This is out of the scope of Pyjnius but we'll show you what the best approach is for playing with it.

Using the Python-for-android project, you can access the default *PythonActivity*. Let's look at an example that demonstrates the *Intent.ACTION_VIEW*:

```
from jnius import cast
from jnius import autoclass

# import the needed Java class
PythonActivity = autoclass('org.kivy.android.PythonActivity')
Intent = autoclass('android.content.Intent')
Uri = autoclass('android.net.Uri')

# create the intent
intent = Intent()
intent.setAction(Intent.ACTION_VIEW)
intent.setData(Uri.parse('http://kivy.org'))

# PythonActivity.mActivity is the instance of the current Activity
# BUT, startActivity is a method from the Activity class, not from our
# PythonActivity.
# We need to cast our class into an activity and use it
currentActivity = cast('android.app.Activity', PythonActivity.mActivity)
currentActivity.startActivity(intent)

# The website will open.
```

3.5 Accelerometer access

The accelerometer is a good example that shows how to write a little Java code that you can access later with Pyjnius.

The SensorManager lets you access the device's sensors. To use it, you need to register a SensorEventListener and overload 2 abstract methods: *onAccuracyChanged* and *onSensorChanged*.

Open your python-for-android distribution, go in the *src* directory, and create a file *org/myapp/Hardware.java*. In this file, you will create everything needed for accessing the accelerometer:

```
package org.myapp;

import org.kivy.android.PythonActivity;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;

public class Hardware {

    // Contain the last event we got from the listener
    static public SensorEvent lastEvent = null;

    // Define a new listener
    static class AccelListener implements SensorEventListener {
        public void onSensorChanged(SensorEvent ev) {
            lastEvent = ev;
        }
        public void onAccuracyChanged(Sensor sensor , int accuracy) {
        }
```

(continues on next page)

(continued from previous page)

```
    }

    // Create our listener
    static AccelListener accelListener = new AccelListener();

    // Method to activate/deactivate the accelerometer service and listener
    static void accelerometerEnable(boolean enable) {
        Context context = (Context) PythonActivity.mActivity;
        SensorManager sm = (SensorManager) context.getSystemService(Context.SENSOR_
↪SERVICE);
        Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

        if (accel == null)
            return;

        if (enable)
            sm.registerListener(accelListener, accel, SensorManager.SENSOR_DELAY_
↪GAME);
        else
            sm.unregisterListener(accelListener, accel);
    }
}
```

So we created one method named *accelerometerEnable* to activate/deactivate the listener. And we saved the last event received in *Hardware.lastEvent*. Now you can use it in Pyjnius:

```
from time import sleep
from jnius import autoclass

Hardware = autoclass('org.myapp.Hardware')

# activate the accelerometer
Hardware.accelerometerEnable(True)

# read it
for i in xrange(20):

    # read the last event
    lastEvent = Hardware.lastEvent

    # we might not get any events.
    if not lastEvent:
        continue

    # show the current values!
    print(lastEvent.values)

    sleep(.1)

# don't forget to deactivate it
Hardware.accelerometerEnable(False)
```

You'll obtain something like this:

```
[-0.0095768067985773087, 9.4235782623291016, 2.2122423648834229]
...
```

3.6 Using TextToSpeech

Same as the audio capture, by looking at the An introduction to Text-To-Speech in Android blog post, it's easy to do it with Pyjnius:

```
from jnius import autoclass
Locale = autoclass('java.util.Locale')
PythonActivity = autoclass('org.kivy.android.PythonActivity')
TextToSpeech = autoclass('android.speech.tts.TextToSpeech')
tts = TextToSpeech(PythonActivity.mActivity, None)

# Play something in english
tts.setLanguage(Locale.US)
tts.speak('Hello World.', TextToSpeech.QUEUE_FLUSH, None)

# Queue something in french
tts.setLanguage(Locale.FRANCE)
tts.speak('Bonjour tout le monde.', TextToSpeech.QUEUE_ADD, None)
```

CHAPTER 4

API

This part of the documentation covers all the interfaces of Pyjnius.

4.1 Reflection classes

class jnius.**JavaClass**

Base for reflecting a Java class, allowing access to that Java class from Python. The idea is to subclass this JavaClass, add few *JavaMethod*, *JavaStaticMethod*, *JavaField*, *JavaStaticField*, and you're done.

You need to define at minimum the *__javaclass__* attribute, and set the *__metaclass__* to `MetaJavaClass`.

So the minimum class definition would look like:

```
from jnius import JavaClass, MetaJavaClass

class Stack(JavaClass):
    __javaclass__ = 'java/util/Stack'
    __metaclass__ = MetaJavaClass
```

__metaclass__

Must be set to `MetaJavaClass`, otherwise, all the methods/fields declared will be not linked to the JavaClass.

Note: Make sure to choose the right metaclass specifier. In Python 2 there is `__metaclass__` class attribute, in Python 3 there is a new syntax `class Stack(JavaClass, metaclass=MetaJavaClass)`.

For more info see PEP 3115.

__javaclass__

Represents the Java class name, in the format 'org/lang/Class' (e.g. 'java/util/Stack'), not 'org.lang.Class'.

__javaconstructor__
If not set, we assume the default constructor takes no parameters. Otherwise, it can be a list of all possible signatures of the constructor. For example, a reflection of the String java class would look like:

```
class String(JavaClass):
    __javaclass__ = 'java/lang/String'
    __metaclass__ = MetaJavaClass
    __javaconstructor__ = (
        '()V',
        '(Ljava/lang/String;)V',
        '([C)V',
        '([CII)V',
        # ...
    )
```

class jnius.**JavaMethod**
Reflection of a Java method.

__init__(*signature*, *static=False*)
Create a reflection of a Java method. The signature is in the JNI format. For example:

```
class Stack(JavaClass):
    __javaclass__ = 'java/util/Stack'
    __metaclass__ = MetaJavaClass

    peek = JavaMethod('()Ljava/lang/Object;')
    empty = JavaMethod('()Z')
```

The name associated with the method is automatically set from the declaration within the JavaClass itself.

The signature can be found with *javap -s*. For example, if you want to fetch the signatures available for *java.util.Stack*:

```
$ javap -s java.util.Stack
Compiled from "Stack.java"
public class java.util.Stack extends java.util.Vector{
public java.util.Stack();
  Signature: ()V
public java.lang.Object push(java.lang.Object);
  Signature: (Ljava/lang/Object;)Ljava/lang/Object;
public synchronized java.lang.Object pop();
  Signature: ()Ljava/lang/Object;
public synchronized java.lang.Object peek();
  Signature: ()Ljava/lang/Object;
public boolean empty();
  Signature: ()Z
public synchronized int search(java.lang.Object);
  Signature: (Ljava/lang/Object;)I
}
```

class jnius.**JavaStaticMethod**
Reflection of a static Java method.

class jnius.**JavaField**
Reflection of a Java field.

__init__(*signature*, *static=False*)
Create a reflection of a Java field. The signature is in the JNI format. For example:

```
class System(JavaClass):
    __javaclass__ = 'java/lang/System'
    __metaclass__ = MetaJavaClass

    out = JavaField('()Ljava/io/InputStream;', static=True)
```

The name associated to the method is automatically set from the declaration within the JavaClass itself.

class jnius.**JavaStaticField**
Reflection of a static Java field.

class jnius.**JavaMultipleMethod**
Reflection of a Java method that can be called from multiple signatures. For example, the method *getBytes* in the *String* class can be called from:

```
public byte[] getBytes(java.lang.String)
public byte[] getBytes(java.nio.charset.Charset)
public byte[] getBytes()
```

Let's see how you could declare that method:

```
class String(JavaClass):
    __javaclass__ = 'java/lang/String'
    __metaclass__ = MetaJavaClass

    getBytes = JavaMultipleMethod([
        '(Ljava/lang/String;)[B',
        '(Ljava/nio/charset/Charset;)[B',
        '()[B'])
```

Then, when you try to access this method, it will choose the best method available according to the type of the arguments you're using. Internally, we calculate a "match" score for each available signature, and take the best one. Without going into the details, the score calculation looks something like:

- a direct type match is +10
- a indirect type match (like using a *float* for an *int* argument) is +5
- object with unknown type (`JavaObject`) is +1
- otherwise, it's considered as an error case, and returns -1

4.2 Reflection functions

jnius.**autoclass**(*name*, *include_protected=True*, *include_private=True*)
Return a *JavaClass* that represents the class passed from *name*. The name must be written in the format *a.b.c*, not *a/b/c*.

By default, autoclass will include all fields and methods at all levels of the inheritance hierarchy. Use the *include_protected* and *include_private* parameters to limit visibility.

```
>>> from jnius import autoclass
>>> autoclass('java.lang.System')
<class 'jnius.reflect.java.lang.System'>
```

autoclass can also represent a nested Java class:

```
>>> autoclass('android.provider.Settings$Secure')
<class 'jnius.reflect.android.provider.Settings$Secure'>
```

Note: If a field and a method have the same name, the field will take precedence.

Note: There are sometimes cases when a Java class contains a member that is a Python keyword (such as *from*, *class*, etc). You will need to use *getattr()* to access the member and then you will be able to call it:

```
from jnius import autoclass
func_from = getattr(autoclass('some.java.Class'), 'from')
func_from()
```

There is also a special case for a *SomeClass.class* class literal which you will find either as a result of *SomeClass.getClass()* or in the *__javaclass__* python attribute.

Warning: Currently *SomeClass.getClass()* returns a different Python object, therefore to safely compare whether something is the same class in Java use *A.hashCode() == B.hashCode()*.

4.3 Java class implementation in Python

class jnius.**PythonJavaClass**

Base for creating a Java class from a Python class. This allows us to implement java interfaces completely in Python, and pass such a Python object back to Java.

In reality, you'll create a Python class that mimics the list of declared *__javainterfaces__*. When you give an instance of this class to Java, Java will just accept it and call the interface methods as declared. Under the hood, we are catching the call, and redirecting it to use your declared Python method.

Your class will act as a Proxy to the Java interfaces.

You need to define at minimum the *__javainterfaces__* attribute, and declare java methods with the *java_method()* decorator.

Note: Static methods and static fields are not supported.

You can only implement Java interfaces. You cannot sub-class a java object.

You must retain a reference to the Python object for the entire liftime that your object is in-use within java.

For example, you could implement the *java/util/ListIterator* interface in Python like this:

```
from jnius import PythonJavaClass, java_method

class PythonListIterator(PythonJavaClass):
    __javainterfaces__ = ['java/util/ListIterator']

    def __init__(self, collection, index=0):
        super(PythonListIterator, self).__init__()
        self.collection = collection
```

(continues on next page)

(continued from previous page)

```
        self.index = index

    @java_method('()Z')
    def hasNext(self):
        return self.index < len(self.collection.data) - 1

    @java_method('()Ljava/lang/Object;')
    def next(self):
        obj = self.collection.data[self.index]
        self.index += 1
        return obj

    # etc...
```

__javainterfaces__
List of the Java interfaces you want to proxify, in the format 'org/lang/Class' (e.g. 'java/util/Iterator'), not 'org.lang.Class'.

__javacontext__
Indicate which class loader to use, 'system' or 'app'. The default is 'system'.

- By default, we assume that you are going to implement a Java interface declared in the Java API. It will use the 'system' class loader.
- On android, all the java interfaces that you ship within the APK are not accessible with the system class loader, but with the application thread class loader. So if you wish to implement a class from an interface you've done in your app, use 'app'.

jnius.**java_method**(*java_signature*, *name=None*)
Decoration function to use with *PythonJavaClass*. The *java_signature* must match the wanted signature of the interface. The *name* of the method will be the name of the Python method by default. You can still force it, in case of multiple signature with the same Java method name.

For example:

```
class PythonListIterator(PythonJavaClass):
    __javainterfaces__ = ['java/util/ListIterator']

    @java_method('()Ljava/lang/Object;')
    def next(self):
        obj = self.collection.data[self.index]
        self.index += 1
        return obj
```

Another example with the same Java method name, but 2 differents signatures:

```
class TestImplem(PythonJavaClass):
    __javainterfaces__ = ['java/util/List']

    @java_method('()Ljava/util/ListIterator;')
    def listIterator(self):
        return PythonListIterator(self)

    @java_method('(I)Ljava/util/ListIterator;',
                 name='ListIterator')
    def listIteratorWithIndex(self, index):
        return PythonListIterator(self, index)
```

4.4 Java signature format

Java signatures have a special format that could be difficult to understand at first. Let's look at the details. A signature is in the format:

```
(<argument1><argument2><...>)<return type>
```

All the types for any part of the signature can be one of:

- L<java class>; = represent a Java object of the type <java class>
- Z = represent a java/lang/Boolean;
- B = represent a java/lang/Byte;
- C = represent a java/lang/Character;
- S = represent a java/lang/Short;
- I = represent a java/lang/Integer;
- J = represent a java/lang/Long;
- F = represent a java/lang/Float;
- D = represent a java/lang/Double;
- V = represent void, available only for the return type

All the types can have the *[* prefix to indicate an array. The return type can be *V* or empty.

A signature like:

```
(ILjava/util/List;)V
-> argument 1 is an integer
-> argument 2 is a java.util.List object
-> the method doesn't return anything.

(java.util.Collection;[java.lang.Object;)V
-> argument 1 is a Collection
-> argument 2 is an array of Object
-> nothing is returned

([B)Z
-> argument 1 is a Byte []
-> a boolean is returned
```

When you implement Java in Python, the signature of the Java method must match. Java provides a tool named *javap* to get the signature of any java class. For example:

```
$ javap -s java.util.Iterator
Compiled from "Iterator.java"
public interface java.util.Iterator{
public abstract boolean hasNext();
  Signature: ()Z
public abstract java.lang.Object next();
  Signature: ()Ljava/lang/Object;
public abstract void remove();
  Signature: ()V
}
```

The signature for methods of any android class can be easily seen by following these steps:

```
1. $ cd path/to/android/sdk/
2. $ cd platforms/android-xx/  # Replace xx with your android version
3. $ javap -s -classpath android.jar android.app.Activity  # Replace android.app.
↪Activity with any android class whose methods' signature you want to see
```

4.5 Java Lambda implementation in Python using Lambdas and Function References

It is possible to use Python lambdas or function references to implement Java *functional interfaces <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html#package.description>*. A functional interface has one (non-default) method. When implementing a functional interface in Python, your lambda must have the correct number of parameters and return the correct data type. You must hold a reference to the Python lambda for as long as it will be used by Java.

For example, here we use a Python lambda to implement the *Comparator <https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>* functional interface:

```
numbers = autoclass('java.util.ArrayList')()
Collections = autoclass('java.util.Collections')
numbers.add(1)
numbers.add(3)
revSort = lambda i, j: j - i
Collections.sort(numbers, revSort)
```

The lambda is wrapped in a PythonJavaClass, which implements the Java interface of the parameter in the called Java method.

Passing Variables: By Reference or By Value

When Python objects such as *lists* or *bytearrays* are passed to Java Functions, they are converted to Java arrays. Since Python does not share the same memory space as the JVM, a copy of the data needs to be made to pass the data.

Consider that the Java method might change values in the Java array. If the Java method had been called from another Java method, the other Java method would see the value changes because the parameters are passed by reference. The two methods share the same memory space. Only one copy of the array data exists.

In Pyjnius, Python calls to Java methods simulate pass by reference by copying the variable values from the JVM back to Python. This extra copying will have a performance impact for large data structures. To skip the extra copy and pass by value, use the named parameter *pass_by_reference*.

```
obj.method(param1, param2, param3, pass_by_reference=False)
```

Since Java does not have function named parameters like Python does, they are interpreted by Pyjnius and are not passed to the Java method.

In the above example, the *pass_by_reference* parameter will apply to all the parameters. For more control you can pass a *list* or *tuple* instead.

```
obj.method(param1, param2, param3, pass_by_reference=(False, True, False))
```

If the passed *list* or *tuple* is too short, the final value in the series is used for the remaining parameters.

4.6 JVM options and the class path

JVM options need to be set before *import jnius* is called, as they cannot be changed after the VM starts up. To this end, you can:

```
import jnius_config
jnius_config.add_options('-Xrs', '-Xmx4096')
jnius_config.set_classpath('.', '/usr/local/fem/plugins/*')
import jnius
```

If a classpath is set with these functions, it overrides any CLASSPATH environment variable. Multiple options or path entries should be supplied as multiple arguments to the *add_* and *set_* functions. If no classpath is provided and CLASSPATH is not set, the path defaults to '.'. This functionality is not available on Android.

4.7 Pyjnius and threads

`jnius.detach()`
: Each time you create a native thread in Python and use Pyjnius, any call to Pyjnius methods will force attachment of the native thread to the current JVM. But you must detach it before leaving the thread, and Pyjnius cannot do it for you.

Pyjnius automatically calls this *detach()* function for you when a python thread exits. This is done by monkey-patching the default *run()* method of *threading.Thread* class.

So if you entirely override *run()* from your own subclass of Thread, you must call *detach()* yourself on any kind of termination.

Example:

```
import threading
import jnius

class MyThread(threading.Thread):

    def run(...):
        try:
            # use pyjnius here
        finally:
            jnius.detach()
```

If you don't, it will crash on dalvik and ART / Android:

```
D/dalvikvm(16696): threadid=12: thread exiting, not yet detached (count=0)
D/dalvikvm(16696): threadid=12: thread exiting, not yet detached (count=1)
E/dalvikvm(16696): threadid=12: native thread exited without detaching
E/dalvikvm(16696): VM aborting
```

Or:

```
W/art     (21168): Native thread exiting without having called DetachCurrentThread␣
↪(maybe it's going to use a pthread_key_create destructor?): Thread[16,tid=21293,
↪Native,Thread*=0x4c25c040,peer=0x677eaa70,"Thread-16219"]
F/art     (21168): art/runtime/thread.cc:903] Native thread exited without calling␣
↪DetachCurrentThread: Thread[16,tid=21293,Native,Thread*=0x4c25c040,peer=0x677eaa70,
↪"Thread-16219"]
```

(continues on next page)

(continued from previous page)

```
F/art     (21168): art/runtime/runtime.cc:203] Runtime aborting...
F/art     (21168): art/runtime/runtime.cc:203] (Aborting thread was not attached to␣
↪runtime!)
F/art     (21168): art/runtime/runtime.cc:203] Dumping all threads without␣
↪appropriate locks held: thread list lock mutator lock
F/art     (21168): art/runtime/runtime.cc:203] All threads:
F/art     (21168): art/runtime/runtime.cc:203] DALVIK THREADS (16):
...
```

CHAPTER 5

Packaging

For Packaging we use PyInstaller and with these simple steps we will create a simple executable containing PyJNIus that prints the path of currently used Java. These steps assume you have a supported version of Python for PyJNIus and PyInstaller available together with Java installed (necessary for running the application).

5.1 main.py

```
from jnius import autoclass

if __name__ == '__main__':
    print(autoclass('java.lang.System').getProperty('java.home'))
```

This will be our `main.py` file. You can now call PyInstaller to create a basic `.spec` file which contains basic instructions for PyInstaller with:

```
pyinstaller main.py
```

5.2 main.spec

The created `.spec` file might look like this:

```
# -*- mode: python -*-

block_cipher = None


a = Analysis(
    ['main.py'],
    pathex=['<some path to main.py folder>'],
    binaries=None,
```

(continues on next page)

(continued from previous page)

```
    datas=None,
    hiddenimports=[],
    hookspath=[],
    runtime_hooks=[],
    excludes=[],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher
)

pyz = PYZ(
    a.pure,
    a.zipped_data,
    cipher=block_cipher
)

exe = EXE(
    pyz,
    a.scripts,
    exclude_binaries=True,
    name='main',
    debug=False,
    strip=False,
    upx=True,
    console=True
)

coll = COLLECT(
    exe,
    a.binaries,
    a.zipfiles,
    a.datas,
    strip=False,
    upx=True,
    name='main'
)
```

Notice the `Analysis` section, it contains details for what Python related files to collect e.g. the `main.py` file. For PyJNIus to work you need to include the `jnius_config` module to the `hiddenimports` list, otherwise you will get a `ImportError: No module named jnius_config`:

```
...

a = Analysis(
    ['main.py'],
    pathex=['<some path to main.py folder>'],
    binaries=None,
    datas=None,
    hiddenimports=['jnius_config'],
    hookspath=[],
    runtime_hooks=[],
    excludes=[],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher
)
```

(continues on next page)

(continued from previous page)

```
...
```

After the `.spec` file is ready, in our case it's by default called by the name of the `.py` file, we need to direct PyInstaller to use that file:

```
pyinstaller main.spec
```

This will create a folder with all required `.dll` and `.pyd` or `.so` shared libraries and other necessary files for our application and for Python itself.

5.3 Running

We have the application ready, but the "problem" is PyJNIus doesn't detect any installed Java on your computer (yet). Therefore if you try to run the application, it'll crash with a `ImportError: DLL load failed: ...`. For this simple example if you can see `jnius.jnius.pyd` or `jnius.jnius.so` in the final folder with `main.exe` (or just `main`), the error indicates that the application could not find Java Virtual Machine.

The Java Virtual Machine is in simple terms said another necessary shared library your application needs to load (`jvm.dll` or `libjvm.so`).

On Windows this file might be in a folder similar to this:

```
C:\Program Files\Java\jdk1.7.0_79\jre\bin\server
```

and you need to include the folder to the system `PATH` environment variable with this command:

```
set PATH=%PATH%;C:\\Program Files\\Java\\jdk1.7.0_79\\jre\\bin\\server
```

After the `jvm.dll` or `libjvm.so` becomes available, you can safely try to run your application:

```
main.exe
```

and you should get an output similar to this:

```
C:\Program Files\Java\jdk1.7.0_79\jre
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

A

D

J

P